# Dissecting Performance of Production QUIC

Alexander Yu*
Brown University
alexander_yu@brown.edu

Theophilus A. Benson
Brown University
tab@cs.brown.edu

## ABSTRACT

IETF QUIC, the standardized version of Google's UDP-based layer-4 network protocol, has seen increasing adoption from large Internet companies for its benefits over TCP. Yet despite its rapid adoption, performance analysis of QUIC in production is scarce. Most existing analyses have only used unoptimized open-source QUIC servers on non-tuned kernels: these analyses are unrepresentative of production deployments which raises the question of whether QUIC actually outperforms TCP in practice.

In this paper, we conduct one of the first comparative studies on the performance of QUIC and TCP against production endpoints hosted by Google, Facebook, and Cloudflare under various dimensions: network conditions, workloads, and client implementations.

To understand our results, we create a tool to systematically visualize the root causes of performance differences between the two protocols. Using our tool we make several key observations. First, while QUIC has some inherent advantages over TCP, such as worst-case 1-RTT handshakes, its overall performance is largely determined by the server's choice of congestion-control algorithm and the robustness of its congestion-control implementation under edge-case network scenarios. Second, we find that some QUIC clients require non-trivial configuration tuning in order to achieve optimal performance. Lastly, we demonstrate that QUIC's removal of head-of-line (HOL) blocking has little impact on web-page performance in practice. Taken together, our observations illustrate the fact that QUIC's performance is inherently tied to implementation design choices, bugs, and configurations which implies that QUIC measurements are not always a reflection of the protocol and often do not generalize across deployments.

## CCS CONCEPTS

• **Networks** → **Protocol testing and verification**.

## KEYWORDS

QUIC, HTTP/3, Transport Protocol, Multiplexing, Congestion Control

**ACM Reference Format:**
Alexander Yu and Theophilus A. Benson. 2021. Dissecting Performance of Production QUIC. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3450103

*Work was done prior to joining Amazon.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.
*WWW '21, April 19–23, 2021, Ljubljana, Slovenia*
© 2021 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC-BY 4.0 License.
ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3450103

## 1 INTRODUCTION

Due to the growing dependence on online services, e.g., Zoom or Netflix, web performance has become a crucial concern for online service providers and online content providers, such as, Google, Facebook, Amazon, and Netflix. Given its importance, it is no surprise that we have witnessed significant innovations in the design of web protocols, e.g., HTTP2 and HTTP3.

One of the core protocols, QUIC, has gained increased adoption and is currently the foundation for emerging protocols, e.g., HTTP3. Given QUIC's importance, there has been tremendous effort to analyze and benchmark its performance [10, 17, 23, 39, 42, 45, 50, 52]. Unfortunately, much of this work either focuses on unoptimized deployments [10, 45, 46] or explores a limited set of payloads [42, 50, 52]. There is a need for more practical and realistic tools for benchmarking QUIC. This need is growing increasingly important as we witness an explosion in the number of distinct and different QUIC implementations [3]. Unfortunately, the lack of general tools which interoperate between different deployments and clients has led to contradictory results and claims between researchers with some stating that QUIC outperforms TCP and others stating the exact opposite [14, 16, 25, 40, 53].

In this paper, our goal is to develop a simple and lightweight but general tool for benchmarking practical QUIC deployments and identifying implementation idiosyncrasies which lies at the cause of performance problems. Our approach builds on the following core principles: First, rather than explore open-source QUIC implementations which are largely unoptimized, we analyze production endpoints of popular web-services, e.g., Google and Facebook. These endpoints allow us to take advantage of closed-source code, configuration, and kernel optimizations. Second, to capture a more holistic view of QUIC performance, we not only compare server implementations but also compare and analyze multiple QUIC clients.

To support these principles, we develop a testing harness that emulates different network conditions, thus enabling us to analyze implementation behavior across a broad range of scenarios. In addition, we design diagnosis scripts that analyze protocol behavior such as bytes acknowledged over time in order to facilitate identification of the root-cause of problems.

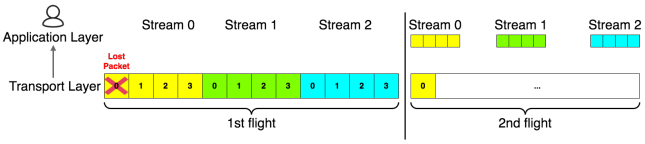
Taken together, our approach reflects a fundamental shift in how QUIC is benchmarked and analyzed. We focus on production deployments, employ heterogeneous client-side implementations, and provide open-source scripts for root-cause diagnosis. This approach enabled us to identify the non-trivial impact of client side implementations on performance. More importantly, it also enabled us to identify problems in Facebook's implementation and work with their team in fixing them.

To illustrate the strength and versatility of our tool, we use it to analyze three production QUIC deployments and three distinct

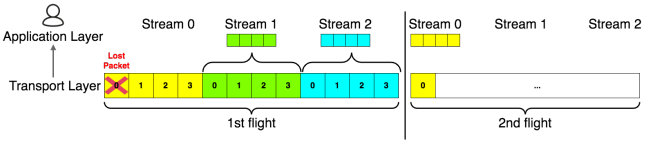
QUIC clients. In analyzing these implementations, we make the following observations:

- QUIC's inherent 1 RTT advantage over TCP for secure connection establishment gives QUIC an edge for small payloads.
- For larger payloads, the server's congestion control has a much larger impact on performance rather than QUIC's design itself.
- Separate packet number spaces, which is QUIC's unique mechanism to logically separate packets with different encryption levels, introduces novel edge-cases that can severely impact congestion control behavior.
- QUIC clients are not necessarily configured with the same TLS options, which leads to performance issues when QUIC servers are configured with specific clients in mind.
- QUIC's removal of HOL (head-of-line) blocking has little impact on Page Load Time and Speed Index relative to congestion control for real-world web-pages.

Our observations show that care must be taken to account for implementation and operational (i.e., configuration) choices when benchmarking and analyzing QUIC. In particular, our study identifies many edge-cases where implementations perform poorly due to non-optimal configuration or coding bugs rather than protocol design. As a result, we demonstrate that even for large content providers, significant engineering efforts must still be made to make QUIC as robust as TCP.




**(a) TCP: Application layer can only read data from streams 1 and 2 until the 2nd flight of packets arrive since TCP has no built-in stream-multiplexing and treats all packets as part of a single 'stream'. Thus, the lost packet at the 'head of the line' is blocking subsequent packets from being processed.**




**(b) QUIC: Application layer can read data from streams 1 and 2 by the 1st flight of packets since QUIC has built-in stream-multiplexing. The lost packet at the 'head of the line' belongs to stream 0, so it does not block in-order packets from streams 1 and 2 from being processed.**

**Figure 1: Head-of-line (HOL) Blocking.**

## 2 QUIC OVERVIEW

QUIC, at its core, is essentially multiplexed TCP and TLS combined into a single transport protocol built on top of UDP. In other words, QUIC provides the same guarantees as TCP, such as reliable delivery of data and strict flow control, while also incorporating security and stream multiplexing directly into its design. In this section, we describe some of QUIC's key differences with TCP and discuss their potential effects on application performance.

**Multiplexed Streams.** Unlike TCP, QUIC incorporates stream multiplexing directly into its transport protocol design. Having this feature loosens the ordering constraints on packet delivery, as data only needs to be delivered in-order at the stream level rather than at the connection level. Consequently, QUIC alleviates TCP's head-of-line (HOL) blocking problem, where a lost packet containing data from one stream prevents the receiver from processing subsequent packets containing data from other streams, which is illustrated in Figure 1. However, the same figure also demonstrates that QUIC's stream multiplexing does not necessarily impact the total time to process all data[1]. As a result, care must be taken to design appropriate experiments to dissect and analyze the impact of QUIC's multiplexing on application performance.

**Connection Establishment.** Another advantage QUIC has over TCP+TLS1.3 is that it requires one less round trip to establish a secure connection for a new or existing session. Unlike QUIC, TCP does not have TLS negotiation built into its protocol, meaning that it must always complete its own handshake, which takes 1 round trip, before initiating the TLS handshake, which also takes 1 round trip. Consequently, because QUIC is designed to use TLS by default, it does not require a separate non-TLS handshake to synchronize the sender and receiver. This means that QUIC should always take at least one less RTT to complete an HTTPS request compared to TCP.

**Loss-recovery.** QUIC's loss-recovery design builds upon decades of TCP deployment experience as it utilizes and simplifies many of TCP's most effective recovery mechanisms, e.g., Fast Retransmit [9], Selective Acknowledgement (TCP SACK) [34], Recent Acknowledgement (TCP RACK) [15], etc. For instance, QUIC introduces monotonically-increasing packet numbers in order to differentiate between fresh and retransmitted data, which greatly simplifies RTT measurements. Overall, QUIC's recovery mechanisms should ensure that it handles loss as well as, if not better than, a state-of-the-art TCP stack.

**HTTP/3 (H3).** QUIC also enables a new application protocol, HTTP/3, to take full advantage of QUIC's stream-multiplexing capabilities. H3 differs from its predecessor, HTTP/2 (H2), in that it removes stream metadata from its header, uses a new header compression algorithm, and adopts a new prioritization scheme [47]. Combined, these features should improve QUIC's performance as they reduce H3's payload size compared to that of H2 and make resource prioritization under H3 more accessible.

**User-space.** QUIC's user-space nature presents unique opportunities in terms of configuration and experimentation at the transport layer. Using QUIC allows implementers to easily test and deploy a variety of flow-control and congestion-control strategies to fit their needs. However, such heterogeneity can also translate to significant performance discrepancies across implementations, creating the need for a broad, general analysis of QUIC performance across independent deployments.

[1] Assuming that both the TCP sender and receiver enable the TCP SACK option.

| Author | QUIC Version | Production end-points | Root Cause Analysis | Results | Explanation |
|---|---|---|---|---|---|
| Google [27] | h3-29 | ✓ | ✗ | QUIC improved client desktop throughput by 3%, decreased search latency by 2%, and decreased video rebuffer rates by 9%. | None provided. |
| Facebook [35] | N/A[2] | ✓ | ✗ | QUIC reduced request errors by 6%, reduced tail latency by 20%, and reduced mean-time-between-rebuffering (MTBR) by 22%. | QUIC's state-of-the-art recovery mechanisms improves tail metrics. |
| Cloudflare [50] | h3-27 | ✓ | ✗ | H3's performance was 1-4% worse than H2's across multiple POP locations. | Different congestion control used for TCP and QUIC. |
| Saif [45] | h3-27 | ✗ | ✗ | In a local environment, QUIC performed worse for all network conditions except high loss. | Code churn and unoptimized open-source implementation. |
| Codavel [10] | h3-20 | ✗ | ✗ | During packet loss, QUIC performed better for small payloads (250KB) but substantially worse for large payloads (17MB) | Unoptimized open-source implementation and different congestion control. |
| This work | h3-29 | ✓ | ✓ | QUIC performed favorably for Google under all network scenarios but had mixed results for Facebook and Cloudflare. | QUIC's performance is largely determined by its congestion control implementation. |

**Table 1: Existing QUIC (HTTP3) vs. TCP (HTTP2) Benchmarks.**

## 3 RELATED WORKS

Most related work [12, 14, 25–27, 41, 44, 53] on QUIC performance focuses on gQUIC (Google's version of QUIC) which is significantly different from the subsequent IETF version of QUIC [49]. These differences imply that the conclusions drawn from previous works are not applicable to a majority of today's QUIC. In short, limitations of these approaches motivate a need for a more up-to-date study.

The standard approach to experiment design has focused on using Chromium's open-source gQUIC server in a local environment. With this approach, experimenters have full control over various dimensions that impact performance, e.g., payload distribution and network conditions. However, they are also responsible for tuning kernel and application settings. Consequently, configuration differences between studies have led to conflicting claims between researchers with some stating that gQUIC outperforms TCP and others stating the exact opposite [14, 16, 25, 40, 44, 53]. These issues highlight the need for general approaches which can benchmark production deployments.

In terms of root cause analysis, these studies cite gQUIC's 0-RTT connection establishment [12, 25], removal of HOL blocking [26, 53], congestion control [14, 25], and improved recovery mechanisms [16, 25, 27] as the main factors behind gQUIC and TCP's different performance outcomes. Our study covers some of these aspects but differs by separating the QUIC protocol from the implementation.

Differentiating between protocol and implementation was not necessary for prior gQUIC studies since only Google's implementation was serving substantial user traffic at the time [29]. With the recent explosion in QUIC implementations [3], brought on by many content providers' decision to develop in-house solutions, there is now a need to compare across these versions in order to gain a more holistic view of QUIC.

The lack of variety among gQUIC implementations has actually led to misleading root cause analysis as well. For instance, when discussing congestion control, researchers have often used the phrase, "QUIC's congestion control..." [12, 14, 25], which is inaccurate considering that there is no standard congestion control for QUIC. In our work, we even show that the same congestion control algorithm on separate QUIC stacks can lead to vastly different performance outcomes based on their implementation quality. With this in mind, our approach allows us to highlight key implementation aspects that lead to different performance outcomes across production deployments.

Along the topic of production deployments, more general studies by content providers have identified various metrics that QUIC improves upon, which we highlight in Table 1. However, none of these content providers have shared any root cause analysis for QUIC's improvements over TCP, leaving those outside these companies with little information on how such improvements were achieved. Furthermore, this absence of detailed analysis makes it difficult to explain the lack of cohesion between their conclusions. It is precisely this problem of explaining QUIC's performance heterogeneity that we address with our more general approach using client-based tools.

## 4 METHODOLOGY

In this section, we describe our methodology for benchmarking QUIC and TCP. Given our emphasis of analyzing production environments, we are only able to benchmark these protocols by selecting appropriate application protocols. Specifically, to benchmark QUIC, we analyzed servers using the HTTP/3 (H3) protocol (which runs atop QUIC) and to benchmark TCP, we analyzed the

[2]Facebook did not share a specific QUIC version in their blog post but they have deployed IETF QUIC since at least h3-9 [23].

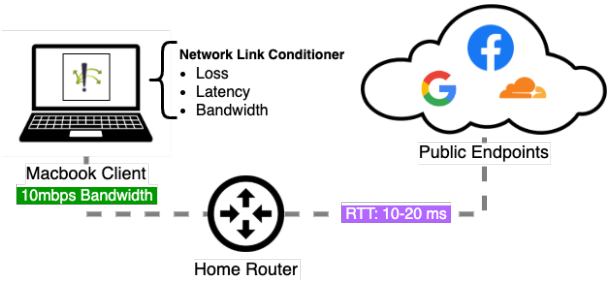



**Figure 2: Testbed setup.**

| Parameter | Values Tested |
|---|---|
| Bandwidth | 10mbps |
| Extra loss | 0%, 0.1%, 1% |
| Extra delay (RTT) | 50ms, 100ms |
| Single-object sizes | 100KB, 1MB, 5MB |
| Web-page sizes | small (≤ 0.47MB)<br>medium (≤ 1.54MB)<br>large (≤ 2.83MB) |
| Content Providers | Facebook, Google, Cloudflare |

**Table 2: Scenarios Tested.**

HTTP/2 (H2) protocol (which runs atop TCP). The benchmarking tools we used for our experiments are open-sourced and can be found at https://github.com/triplewy/quic-benchmarks.

## 4.1 Testbed

*4.1.1* ***Server-side Setup.*** For the server-side, we used publicly-available endpoints from Google, Facebook, and Cloudflare. To the best of our knowledge, these were the only companies that served QUIC in production as of December 2020. We tested QUIC version 29, which will closely resemble the final specification considering that changes made since have had little impact on the overall protocol [21]. Thus, our benchmark results and analysis will be applicable for future QUIC versions.

We benchmarked two types of workloads: single-object resources, which allowed us to analyze raw protocol behavior, and, multi-object web-pages which allowed us to explore protocol behavior during the page load process. For both types of workloads, we chose static resources (e.g., images and text files) or static pages (e.g., about pages, blog posts) that were accessible without user credentials in order to avoid triggering user-based business logic that could have impacted performance measurements.

*4.1.2* ***Client-side Setup.*** We conducted our experiments using a Macbook Air (OSX 10.13.6, Intel Core i5 1.3 GHz, 8 GB memory) on a home internet connection. The RTT between our client and content providers' endpoints was consistently between 10-20ms, as our router always resolved their domain names to Point-of-Presences (POPs) near Boston.

For our H2 clients, we used Google Chrome and cURL, and for our H3 clients, we used Google Chrome, Facebook Proxygen, and Ngtcp2. We chose Facebook Proxygen and Ngtcp2 as our H3 command-line clients since Proxygen client has been deployed in Facebook mobile apps since at least 2017 [39] and Ngtcp2 is a popular QUIC library used in various open-source projects [1, 48].

In terms flow control configuration, for TCP we used OSX's default settings where TCP buffer size starts at 128KB and increases up to 1MB due to TCP window-size scaling. As for QUIC, whose flow control consists of per-connection and per-stream parameters, we used 15MB at the connection-level and 6MB at the stream-level for Chrome. These particular values represent Chrome's default settings and happen to be larger than any of our benchmark's payloads. For Proxygen and Ngtcp2, we used 1GB for both parameters to also make QUIC flow control was a non-factor in our experiments.

In terms of TCP options, OSX, by default, enables TCP SACK but does not enable ECN. Similarly, none of our QUIC clients enable ECN. Lastly, to keep H2 and H3 comparisons consistent, we disabled SSL verification in cURL since both Proxygen and Ngtcp2 do not support SSL verification in their vanilla implementations. However, Chrome does not provide the option to disable SSL verification, which had an impact on our H3 client consistency results, which we discuss in §7.2.

## 4.2 Network Environments

We used Network Link Conditioner [36] to simulate various network conditions from the client side. Under the hood, Network Link Conditioner uses `dnctl` and `pfctl` which are based on OpenBSD's network shaper and packet filter tools [28]. These tools work similarly to Linux's tc-netem [5] in that packets are first filtered based on layer-3 or layer-4 protocol information, then buffered into a queue where bandwidth, delay, and loss rates are applied [24]. Added loss from Network Link Conditioner is distributed randomly while added delay is distributed uniformly.

Table 2 shows the various network conditions and payloads that we tested. All of our experiments were run at a fixed bandwidth of 10 mbps on a home Wifi connection. Our reasoning behind these specific network conditions is that they represent normal and realistic edge case network conditions, e.g., high loss or high latency, that can induce unique protocol behavior for root cause analysis.

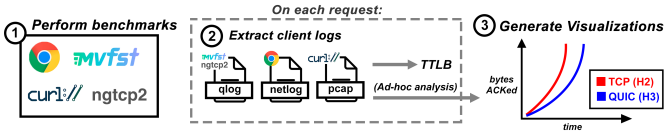
Lastly, considering the inherent network variability associated with benchmarking over the Internet, we ran each experiment at least 40 times to reduce the impact of this network variability on our results.

# 5 EVALUATION FRAMEWORK

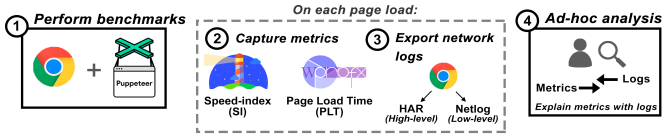
In this section, we describe our performance metrics and our method for determining performance differences.

**Metrics.** For our single-object benchmarks, we measured network performance using time-to-last-byte (TTLB), which we define as the time difference between the first packet sent by the client and the last packet received from the server.

As for our multi-object benchmarks, we measured Speed Index (SI) [7], a metric that indicates how quickly a page is visibly populated, and Page-Load Time (PLT), a commonly used metric for

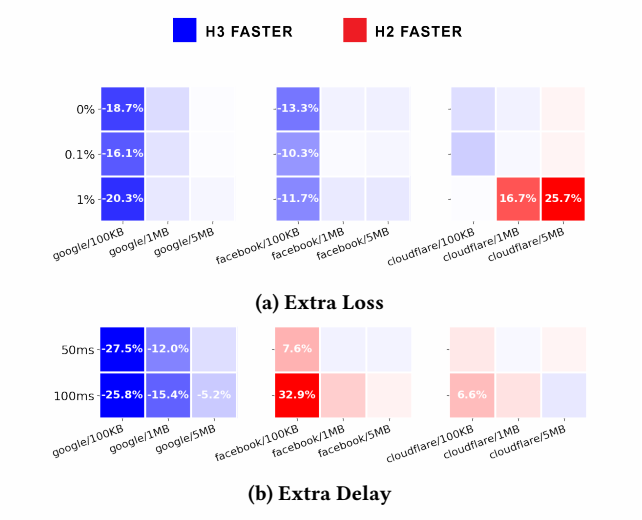



(a) Single-object workflow.




(b) Multi-object workflow.

**Figure 3: Measurement tool workflows.**




(a) Extra Loss

(b) Extra Delay

**Figure 4: H2 versus H3 performance for single-object endpoints during 10mbps bandwidth. Percentage differences less than 5% are not labelled.**

web-page performance [12, 14, 25, 43, 50]. We used Google Lighthouse [4], a tool to perform audits on web-pages, to capture SI from our Chrome client and WProfX [38], a tool that identifies a web-page's critical loading path, to calculate PLT from exported Chrome trace events.

**Statistical Analysis.** Due to the nature of benchmarking on the Internet, we employed median instead of mean when comparing performance. Median's main advantage over mean is that it ignores outlier data points, which are inevitable when performing Internet benchmarks. Such outliers may be caused by various factors unrelated to QUIC or TCP, like resource caching on the server, transient congestion in middleboxes, Wifi interference, etc.

## 6 MEASUREMENT TOOLS

The main goal for our measurement tools is to identify the root causes behind performance discrepancies between production QUIC and TCP. Our emphasis on evaluating and analyzing production performance requires a fundamental shift in approach compared to prior QUIC studies, whose conclusions are mainly based off local experiments [10, 12, 14, 16, 26, 40, 44, 45, 53].

**Single-object workflow.** We first use a broad set of H2 and H3 clients to identify interesting or unexpected performance discrepancies that arise from benchmarking production H2 and H3 endpoints. Once we have a specific combination of object size, network condition, client set, and production deployment that produces interesting results, we use Python scripts to generate request-level visualizations of bytes ACKed over time from client logs. These visualizations help us identify the manner in which performance deviations occur between H2 and H3 requests. If client analysis is insufficient, we then attempt to reproduce the behavior locally in order to analyze server logs which provides us complete information of requests.

Figure 3a illustrates the generalized workflow for our single-object benchmarks and shows that our H2 and H3 clients generate network logs in three formats: Qlog [33] (exported by our command-line H3 clients, Proxygen and Ngtcp2), Netlog [18] (exported by Chrome), and Pcap (captured via tcpdump and used for our H2 clients). All three log formats provide detailed information on sent and received packets, which is sufficient for creating the aforementioned visualizations. Overall, we find that this approach of connecting real-world findings with client analysis or local reproduction bypasses concerns of misconfiguration and provides relevant insight into the aspects of QUIC that are lacking at the production level.

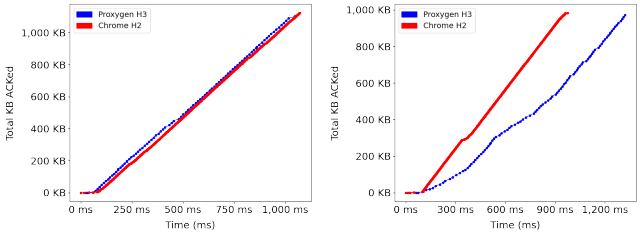
**Multi-object workflow.** Analyzing multi-object web-page performance requires a separate workflow from our single-object analysis since a page-load may consist of tens to hundreds of unique network requests. To this end, the goal of our multi-object measurement tool is to correlate high-level Quality of Experience (QoE) metrics, such as Speed Index and Page Load Time, with low-level network events. Figure 3b shows that we achieve this by using Puppeteer [6] to manipulate Chrome, Lighthouse [4] and WProfX [38] to record QoE metrics, Chrome-HAR [2] to capture high-level network request info, and Chrome Netlog [18] to record low-level network details. We then perform ad-hoc root cause analysis in order to explain the captured metrics with network details from the relevant page loads.

## 7 SINGLE-OBJECT RESULTS

In this section, we present our benchmark results for single-object endpoints and provide root cause analysis for various measured performance differences between QUIC and TCP, and between QUIC clients.

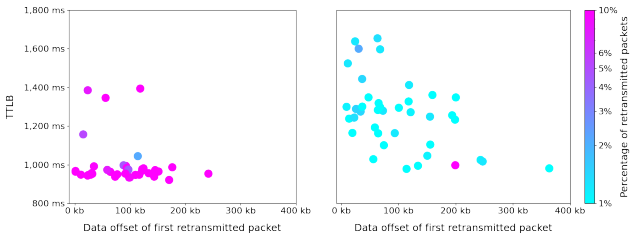
### 7.1 TCP (H2) vs QUIC (H3) Performance

For single-object web resources, we expected the difference between our TCP and QUIC results to be minimal since QUIC only uses one bidirectional stream to send and receive data for a single web object, which is effectively the same as using a TCP connection. However, our results show the contrary since there are major TTLB differences between H2 and H3 for numerous network conditions and object sizes.




(a) Google 1MB Endpoint (b) Cloudflare 1MB Endpoint

**Figure 5: Bytes ACKed over time for 1MB payloads during 1% added loss. Each dot in the graph represents an ACK sent from the client.**



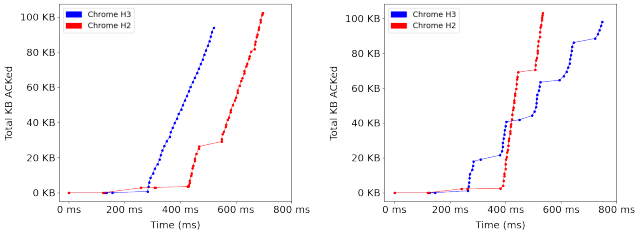

(a) TCP: BBR (b) QUIC: Cubic

**Figure 6: First-retransmission-data-offset vs. TTLB for our 1MB Cloudflare endpoint during 1% loss.**

*7.1.1* ***No Extra Loss or Delay.*** The first row in Figure 4a shows that without added loss or latency, H3 consistently performed better than H2 for 100KB payloads. However, for larger payloads, H3's performance was essentially equivalent to H2's. We ascribe this behavior to QUIC's 1 RTT handshake advantage over TCP for two reasons: First, the behavior is consistent across content providers which implies that protocol-design rather than implementation-design is the root cause. Second, as a matter of logic, a 1 RTT difference has a much larger impact on short-lived connections compared to longer ones. Thus, our data demonstrates that with a network bandwidth of 10mbps, QUIC's 1 RTT advantage has a significant impact on connections whose payload is less than 1MB, but negligible impact otherwise.

*7.1.2* ***Extra Loss.*** For our added loss benchmark results, we see the same pattern of better H3 performance for 100KB payloads but equivalent H3 performance for larger payloads. Cloudflare is the only exception to this pattern, where H3 performed much worse than H2 during 1% added loss. To identify the root cause behind Cloudflare's poor QUIC performance, we first plotted bytes ACKed over time from our median requests to Cloudflare and Google 1MB endpoints during 1% loss in order to visualize the differences in packet behavior between these two QUIC deployments.

Figure 5 shows that for Cloudflare, H3 lagged behind H2 from the onset of their requests and was never able to catch up. This greatly contrasts with Google's behavior where H2 and H3 were virtually identical throughout their requests. Considering that Cloudflare




(a) Google 100KB Endpoint (b) Facebook 100KB Endpoint

**Figure 7: Bytes ACKed over time for 100KB payloads during 100ms added delay.**

uses Cubic for QUIC's congestion control and BBR for TCP's, we suspected that Cubic's handling of early loss was the root cause behind H3's immediate degraded ACK rate. To confirm this, we ran 40 more iterations with our Proxygen H3 and cURL H2 client against Cloudflare's 1MB endpoint during 1% added loss and plotted the relationship between TTLB and data offset of the first retransmitted packet.

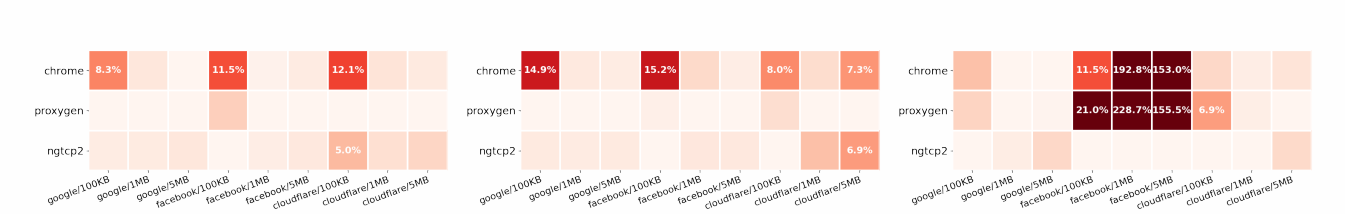
Figure 6 shows a much stronger correlation between TTLB and first-retransmitted-offset for QUIC (Cubic) compared to TCP (BBR). Interestingly, TCP's far greater percentage of retransmissions had little impact on its performance compared to QUIC. This unexpectedly large amount of retransmissions was caused by our TCP buffer being filled before Cloudflare reacted to loss, leading to extra dropped packets. Thus, despite experiencing an order of magnitude more loss than QUIC, TCP was still able to achieve better TTLB, demonstrating how the congestion control's reaction to loss affects TTLB far more than the total number of lost packets.

Cubic is prone to performance degradation during early packet loss since the sender sets $W_{max}$ (Window max) to the current *cwnd* whenever it detects a lost packet. When there is early loss, *cwnd* is bound to be small since it has only received a few ACKs to increase its size exponentially during slow start. Unlike Cubic, BBR is not as severely hindered by early loss since it uses the data acknowledgement rate rather than loss as a signal for congestion. As a result, Cloudflare's poor H3 performance demonstrates Cubic's difficulty in utilizing a link's full bandwidth during random loss.

While sustained, random loss at 10mbps bandwidth may not reflect a common, real world network condition, our analysis still shows that congestion control can significantly impact QUIC performance. Therefore, it is important to differentiate between QUIC and its congestion control as doing so incorrectly leads to false pretenses about QUIC.

*7.1.3* ***Extra Delay.*** For our added delay benchmark results, Figure 4b shows that H3 performed much better than H2 against Google endpoints, but performed significantly worse against Facebook's 100KB endpoint. This was quite surprising given QUIC's aforementioned 1 RTT handshake advantage.

To identify the root cause behind Facebook's poor QUIC performance during added delay, we again underwent the same process described in §7.1.2 where we first used client request logs to plot bytes ACKed over time in order to compare the packet behavior




**(a) 0% Loss** **(b) 1% Loss** **(c) 100ms Delay**

**Figure 8: Various heatmaps showcasing performance differences between H3 clients. Each cell's value is calculated by taking the percent difference between the median TTLB for that client-endpoint tuple and the lowest median TTLB amongst all three clients for that endpoint. Thus, a percent difference of 0% is the best value a client can achieve. Percentage differences less than 5% are not labelled.**

between different endpoints. Figure 7 demonstrates that for both Google and Facebook, Chrome H3 began ACKing application data earlier than Chrome H2, which was due to QUIC's 1 RTT handshake advantage. However, Chrome H3 was unable to maintain this advantage when interacting with Facebook.

Our discussion with a Proxygen maintainer to demystify these surprising results led to uncovering a bug in Proxygen's QUIC BBR congestion controller, which we explain in section §7.2.1. He also mentioned that they use BBR for their TCP congestion control. So, even though Google and Facebook both use BBR in their QUIC and TCP stacks [13], we only measured a large difference in their QUIC behavior.

Google and Facebook's similarity in TCP behavior is most likely due to their common use of Linux kernel's BBR congestion controller. Unlike QUIC, where deploying BBR requires writing it from scratch or porting an open-source implementation, TCP offers the benefits of a battle-tested BBR implementation with one config change in the Linux kernel [51]. Thus, while QUIC's user-space nature offers greater flexibility in terms of congestion control experimentation, optimizing congestion control implementations to perform well under normal and outlier network conditions is evidently not a straightforward task, even for a large content provider like Facebook.

## 7.2 Client Consistency

While previous studies often solely focus on the server's impact on QUIC performance, we also measure performance differences between H3 clients to gauge the client's impact on QUIC performance.

From our results, we find that Chrome consistently performed worse than its counterparts when downloading 100KB payloads, as shown by Figures 8a and 8b. This slowdown was caused by Chrome's SSL verification, which consistently took around 20ms to finish. As previously mentioned, our Proxygen and Ngtcp2 clients do not perform SSL verification. In context, 20ms represents 13.9%, 14.7%, and 7.2% of the best median TTLB for Google, Facebook, and Cloudflare 100KB endpoints during 0% loss respectively, which generally match Chrome's 100KB percentages shown in our results. Therefore, we find that QUIC clients have equivalent performance


**Figure 9: QUIC Packet Number Spaces: Each space (Initial, Handshake, 1RTT) uses its own encryption keys and ACKs can only be registered between packets in the same space. For a new session, a QUIC client and server must progress through these three spaces sequentially in order to ensure secure transfer of data.**

under a home network connection with virtually no loss, consistent sub 20ms RTT, and 10mbps bandwidth.

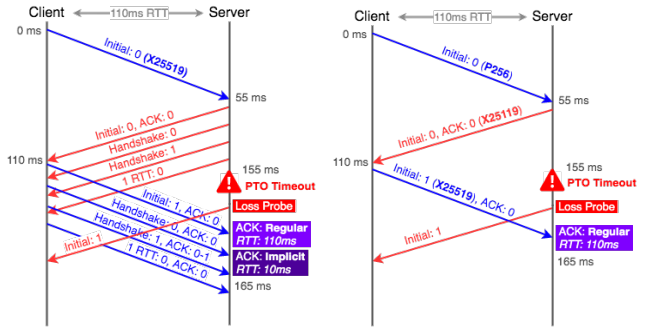
Figure 8c shows more interesting behavior in that Ngtcp2 substantially outperformed its peers when interacting with Facebook endpoints during 100ms added delay. This discrepancy was actually caused by an incompatibility between Ngtcp2's default TLS cipher group and Facebook's TLS setup, which we further discuss in the following section.

*7.2.1* ***Proxygen Implicit ACK Bug.*** Ngtcp2 differs from other clients in that its default TLS cipher group, P256, is incompatible with Facebook Proxygen's default TLS cipher group, X25519. Normally, this results in a 1 RTT disadvantage for Ngtcp2 since it needs to resend its Initial with the correct cipher group after getting Proxygen server's reply. Note that the Initial is the very first packet a client sends to a server as a means to exchange initial encryption keys as shown in Figure 9. However, through local reproduction, we discovered that when adding 100ms delay to the network, this incompatibility actually benefits Ngtpc2 due to a bug in Proxygen's BBR congestion control code.

When RTT is above 100ms, Proxygen server's probe timeout (PTO) will expire before it receives an ACK for its first packet (i.e its Initial packet), causing it to send a probe packet[3]. For clarity, we will subsequently refer to the server's first packet and its probe as $Init_0$ and $Init_1$ since they both belong in the Initial packet number space.

The purpose of a probe packet is to elicit a new ACK from the receiver so that the sender can quickly learn whether the previous

[3]QUIC's PTO is calculated as a function of the connection's RTT, but is set to some arbitrary value when there are no RTT measurements present. For Proxygen, PTO is initially set to 100ms.




**(a) Proxygen/Chrome Client** **(b) Ngctp2 Client**

**Figure 10: QUIC packet sequences with Proxygen server. In both sequences, Proxygen server sends its first packet at timestamp 55ms. At timestamp 155ms, its PTO timeout expires since it has not received a response within 100ms, which triggers a loss probe.**

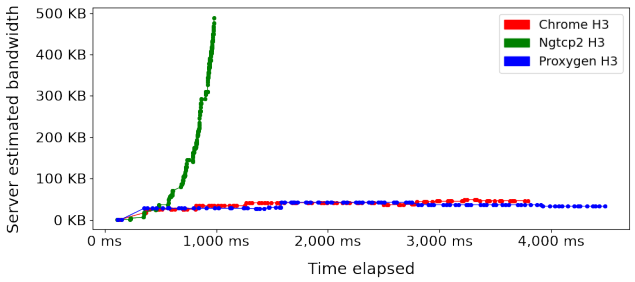
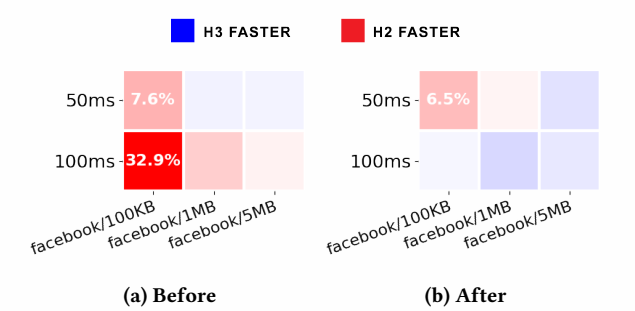



**Figure 11: Local Environment: Server estimated bandwidth for various H3 clients when querying a 1MB payload during 100ms added delay. Each dot in the graph indicates a bandwidth update event from our local Proxygen server.**




**(a) Before** **(b) After**

**Figure 12: Before and after implicit ACK patch: H2 versus H3 performance for single-object Facebook endpoints during added delay. Percent differences less than 5% are not labelled.**

packet was actually lost based on the new ACK information. In addition, the QUIC specification states that a PTO probe must contain new data if available, so in effect, it is treated as a regular packet by the sender. In other words, the PTO will trigger again if the probe itself is not ACKed in time [20].

Figures 10a and 10b illustrate how Proxygen server's PTO expires 100ms after it sends $Init_0$, causing it to send an ACK-eliciting probe, $Init_1$. Then, due to simulated high latency, Proxygen server receives the client's ACK for $Init_0$ immediately after it has sent $Init_1$. The difference between Ngtcp2 and the other H3 clients is that Ngtcp2 responds with a new Initial packet containing new keys rather than Handshake packets.

Once a QUIC server receives a Handshake packet from the client, it must stop processing Initial packets and discard any congestion control state associated with those packets [22]. The purpose of this rule is to allow the server to immediately make forward progress even when there are prior Initial packets yet to be acknowledged. From the server's perspective, receiving a Handshake packet from the client signifies that the Initial phase is completed, meaning that any subsequent Initial packet is irrelevant.

When Proxygen server receives a Handshake packet while $Init_1$ is still in-flight, it 'implicitly' ACKs $Init_1$ so that it can immediately move forward without waiting for its actual ACK. Essentially, an 'implicit' ACK is Proxygen's method of immediately discarding an in-flight, useless Initial packet, which in it of itself is not a problem. The problem is that Proxygen feeds round trip times from 'implicit' ACKs into its congestion control model.

From Figure 10a, we can see that this particular 'implicit' ACK registers an RTT measurement below 10ms when the actual RTT is above 100ms. This has a significant impact on the BBR congestion control model as perceived bandwidth is calculated based on RTT. When BBR encounters such a substantial and sudden increase in RTT, in this case from 10ms to 100ms, it will stop exponentially increasing estimated bandwidth as it perceives that full bottleneck bandwidth has been reached, which is demonstrated in Figure 11.

Ngtcp2, with its default TLS configuration, does not encounter this problem because it does not 'implicitly' ACK the server's PTO probe when it sends a new Initial in the correct format. Later on in the connection, Ngtcp2 normally ACKs the probe packet and thus Proxygen server never encounters any inaccurate RTT measurements.

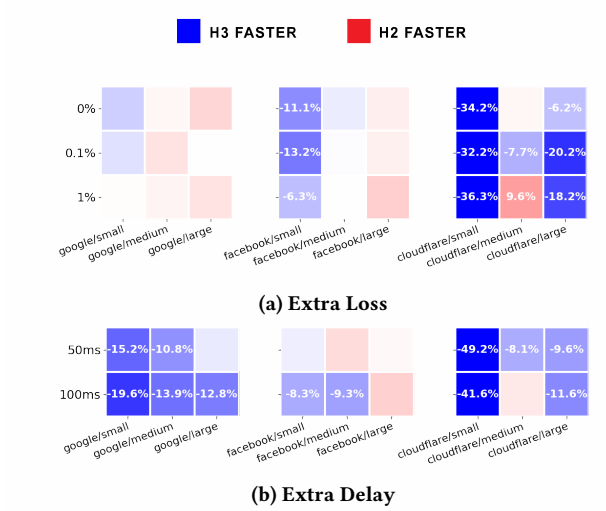
*7.2.2* ***Aftermath.*** The Proxygen maintainers have since implemented a fix that ignores RTT measurements from packets that are 'implicitly' ACKed [8], which is now in line with the QUIC draft specification. After this bug was patched, we reran our benchmark suite against Facebook endpoints during 100ms added delay. Figure 12 shows H3's significant improvement relative to H2 as a result of the patch. Overall, this bug demonstrates how separate packet number spaces, a feature unique to QUIC, creates novel edge cases which can seriously degrade network performance if handled improperly.

This bug also demonstrates that client configuration plays an important role in optimizing QUIC performance. As a result of our analysis, we realized that Ngtcp2 used 2 RTTs instead of 1 RTT to complete its handshakes with Facebook and Cloudflare servers, leading to poor performance relative to other clients. After configuring Ngtcp2 with a compatible TLS group, we found that it performed similarly with other clients and these results are reflected

(a) H2 Page Load - SI: 1.78s

(b) H3 Page Load - SI: 1.70s

**Figure 13: Page-load screenshot sequences of our large-sized Cloudflare web-page during no added loss or delay.**




(a) Extra Loss




(b) Extra Delay

**Figure 14: H2 versus H3 Speed-Index performance for multi-object web-pages during 10mbps bandwidth[4].**

in our client consistency benchmarks in Figure 8 (besides Facebook during 100ms added delay).

## 8 MULTI-OBJECT RESULTS

Barring a few exceptions, our single-object benchmark results have shown that against Google, Facebook, and Cloudflare production endpoints, QUIC performs just as well, if not better than TCP when there is no multiplexing involved. Considering that QUIC is also designed to further improve multiplexing performance with its removal of HOL blocking and its tighter coupling with the application layer, we expected our multi-object results to be even more favorable towards H3 compared to our single-object results. Specifically, we anticipated H3 to show additional performance improvements during added loss where QUIC's removal of HOL blocking is pertinent. However, we expected H3 to perform worse against Cloudflare during added loss due to their use of Cubic for QUIC's congestion control.

[4]Our results are based on using Puppeteer's default browser view-port of 800x600 pixels. We also conducted our multi-object benchmarks following Facebook's 'implicit' ACK patch.

### 8.1 Speed Index

Our Speed Index benchmark results in Figure 14 reject our hypothesis that QUIC's removal of HOL blocking improves web-page performance since H3 did not outperform H2 during added loss for Facebook or Google. While H3 performed well against Cloudflare during added loss, it is evident that web-page layout design rather than protocol design is the root cause since our results are inconsistent across different sized Cloudflare web-pages. Lastly, our results also show that H3 consistently achieved better SI for small-sized web-pages, which again reiterates the positive impact of QUIC's 1-RTT handshake advantage.

In order to fully understand our Speed Index benchmark results, we first explain the calculations behind Speed Index (SI). At its core, SI depicts the amount of change between visual browser frames throughout a page load [7]. A low SI value, which is preferred, indicates that the area visible to the user changes little over time. In other words, immediate loading, as opposed to gradual loading, of prominent visual content translates into a better SI. Note that Lighthouse, the tool we use to capture SI, does not scroll during the loading process, which means that any significant UI content rendered off screen is not factored into SI.

As an example, Figure 13 shows Lighthouse-generated screenshot sequences of our median H2 and H3 page loads for our large-sized Cloudflare web-page. The top sequence (H2) has a worse SI compared to the bottom sequence (H3) since it takes a longer time to load the main image, which is highlighted in red. Relating to our benchmark results in Figure 14, we find that this occurred consistently for our large-sized Cloudflare web-page regardless of the network condition.

*8.1.1* ***Cloudflare.*** In order to pinpoint why H3 consistently outperformed H2 for our large-sized Cloudflare web-page, we first examined the start and end times for the 'main' image resource discussed in the previous section. We define start time and end time as the beginning and end of the main image request and extract this information from Chrome's HAR logs. Figure 15 demonstrates that for each network condition, H3 was able to consistently download this image much faster than H2.

To identify the root cause of this behavior, we used the corresponding netlogs from our median H2 and H3 page loads during 0% loss in order to examine the exact HTTP frames received by the browser. From these HTTP frames, we show in Figure 16 that H2 and H3 differed in their resource load order, where H3 received

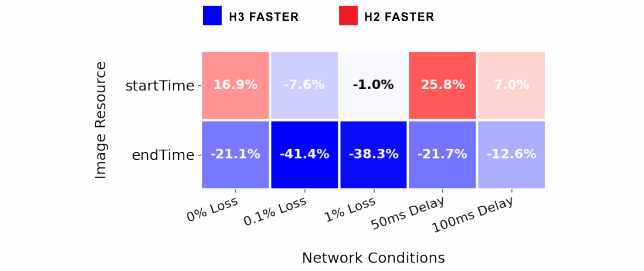



**Figure 15: H2 vs H3 performance for the start and end times of the 'main' image resource of our large-sized Cloudflare web-page.**

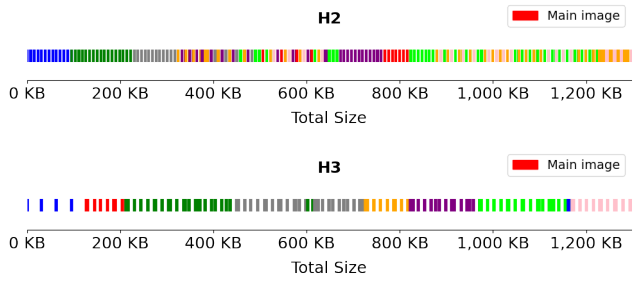



**Figure 16: HTTP frames received by Chrome for 8 simultaneously-requested image/gif resources from our large-sized Cloudflare web-page. Each color represents a unique HTTP stream and each bar represents a chunk of data read by the application layer. Notice that H3 not only received the main image's[5]frames earlier, but also received them exclusively (H2 started receiving main image frames around the 500KB mark).**

the main image's data much earlier despite sending its respective request later, as evidenced by Figure 15's '0% Loss' column.

Since the browser can determine resource download order by attaching prioritization 'hints' to individual resource requests [31], we accordingly examined browser-assigned prioritization values for each resource. We found that H2 and H3 had essentially the same prioritization layout. While H2 and H3 use different prioritization schemes [31], they both specified that the main image should be loaded sequentially before other large image/gif content.

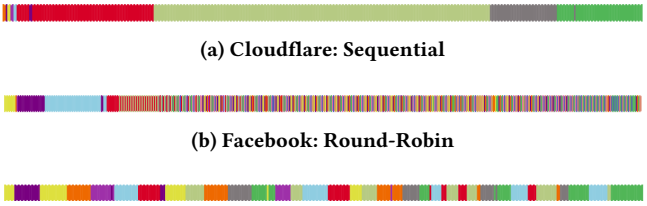
Thus, Cloudflare's edge servers *seemed* to respect H3's prioritization hints but completely disregarded H2's. We asked a Cloudflare engineer about this behavior to which he responded that Cloudflare has implemented features in their H2 stack that ignores the browser's prioritization suggestions for image content [11, 37]. Cloudflare cites performance improvements as a result of these changes, but our benchmark results show that browser prioritization values are still crucial for visual QoE.

The Cloudflare engineer also stated that Cloudflare's H3 stack does not currently support prioritization. So Cloudflare's apparent adherence to Chrome's H3 prioritization values was actually a result of their default FIFO scheduler for multiplexed streams. Essentially, the browser via H3 prioritization indicated that content should be loaded sequentially, which happened to coincide with Cloudflare's default QUIC stream scheduler.

From our analysis, we conclude that H3's performance improvement over H2 for our large-sized Cloudflare web-page was not caused by the QUIC protocol, but instead caused by differences in application configuration. These differences were not mentioned in Cloudflare's discussion of their own H3 benchmarks [50], which demonstrates how our holistic approach helps identify previously omitted, non-protocol factors that skew comparisons between H2 and H3.

*8.1.2* ***Stream Multiplexing.*** Stream multiplexing describes the manner in which a sender transmits data from multiple streams at once. There is a wide spectrum of multiplexing strategies, ranging from sequential, where all bandwidth is allotted to one resource at a time, to round-robin, where each resource gets a slice of bandwidth [32]. Both H2 and H3 support all of these multiplexing strategies. Their only difference is that H2 handles the multiplexing whereas H3 hands that responsibility off to QUIC. Figure 17 shows

[5]'Main image' refers to the image highlighted in Figures 13a and 13b.



**(a) Cloudflare: Sequential**

**(b) Facebook: Round-Robin**

**(c) Google: Batched Round-Robin**

**Figure 17: QUIC stream multiplexing strategies. Each color represents a unique HTTP stream and each segment represents consecutive frames from the same stream.**

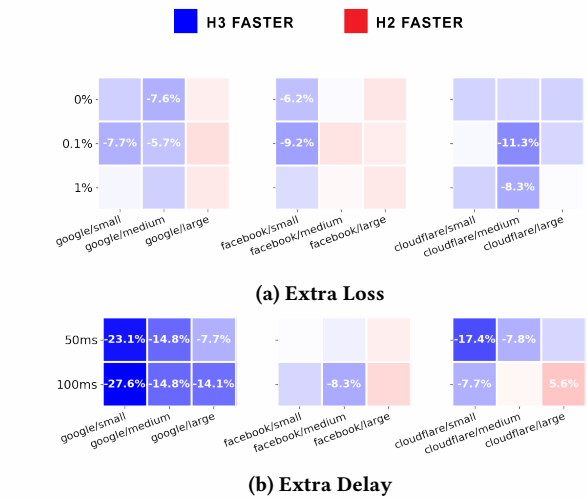
the various QUIC multiplexing strategies deployed in production at Google, Facebook, and Cloudflare.

Cloudflare's use of sequential ordering for QUIC means that Cloudflare's QUIC servers do not multiplex resources in parallel. Consequently, QUIC's removal of HOL blocking does not actually play a role in Cloudflare's multi-object benchmark results. While one might think that switching to round-robin is an obvious upgrade, in practice the advantages of using round-robin are much more nuanced due to resource prioritization, as shown in the previous section, and potential packet loss patterns [30].

Facebook and Google on the other hand use round-robin scheduling. While previous work has shown that round-robin multiplexing in QUIC decreases the amount of data blocked in the transport layer compared to TCP [30–32], our multi-object experiments with added loss show that its effect on SI is negligible. However, we only cover randomly distributed loss so other loss distributions, such as burst loss, may lead to different outcomes.

## 8.2 Page Load Time

The ramifications of our PLT results differ from our SI results since PLT measures a single point in time (when the browser triggers




(a) Extra Loss

(b) Extra Delay

**Figure 18: H2 versus H3 PLT performance for multi-object web-pages during 10mbps bandwidth.**

the page load event) rather than a delta over time. Consequently, resource load order does not impact PLT as much as SI. Still, our PLT results generally align with our SI results as Figure 18 shows that H3 has a slight advantage over H2 for small web-pages but no clear advantage for larger web-pages. Our Cloudflare results are more interesting in that they show H3 is again better than or equivalent to H2 during added loss, which again contrasts with our single-object benchmark results in §7.1.2.

One potential factor that likely contributed to the disconnect between our single object and multi-object Cloudflare results is the fact that our Cloudflare web-pages use many 3rd-party content that can only be served over TCP (H2 or H1.1)[6]. In contrast, Facebook and Google were able to fully serve H3 content from their web-pages. Table 3 shows that JavaScript (JS) and CSS files were the most common types of content to be served over TCP. Since JavaScript execution and CSS rendering impacts the page load process [19], this prevalence of 3rd party content served over TCP likely impacted our PLT results.

Overall, besides Cloudflare, our PLT results are quite similar to our single-object results, demonstrating that congestion control and QUIC's lower latency handshakes have far more impact on PLT than QUIC's removal of HOL blocking. Furthermore, when considering the presence of 3rd-party TCP content in Cloudflare web-pages and the aforementioned differences between Cloudflare's H2 and H3 multiplexing strategies, it is difficult for us to accurately measure the impact of protocol design on our Cloudflare PLT results.

## 9 CONCLUSION

In this paper, we present a novel approach for evaluating QUIC's performance that focuses on benchmarking and analyzing various production deployments. Previous studies have mainly evaluated QUIC in local environments, which has led to conflicting claims on the protocol. By replacing local experimentation with production

[6]During our web-page selection process, we could not find any static Cloudflare web-page that only requested H3 content.

| Page size | HTML | CSS | JS | Image/GIF |
|---|---|---|---|---|
| Small | 100% | 100% | 98% | N/A |
| Medium | 94% | 97% | 75% | 99% |
| Large | 44% | 29% | 45% | 98% |

**Table 3: Cloudflare web-pages: Percentage of bytes transferred using H3 categorized by resource type.**

experimentation, we bypass the concern of non-optimal environment or application configuration. As a result, we have identified multiple protocol and implementation aspects that cause performance discrepancies between QUIC and TCP and between QUIC implementations.

Our findings demonstrate that most performance differences can be attributed to developer design and operator configuration choices, e.g., selection of congestion control algorithms or cipher suites. In most cases, performance differences were due to implementation details and not due to specific inherent properties of the QUIC protocol.

With this in mind, we conclude that optimizing QUIC in practice can be difficult and time-consuming considering the overhead of dealing with QUIC edge cases and implementing congestion control algorithms from scratch. We show that QUIC has inherent advantages over TCP in terms of its latency, versatility, and application-layer simplicity. However, these benefits come at the cost of high implementation overhead, leading to inconsistencies in performance across implementations. By demonstrating large differences between Google, Facebook, and Cloudflare's QUIC performance profiles, we show that deploying QUIC does not automatically lead to improvements in network or application performance for many use cases.

## ACKNOWLEDGMENTS

We'd like to thank the anonymous reviewers, Robin Marx, Matt Joras, Lucas Pardue, Usama Naseer, Luca Niccolini, Tatsuhiro Tsujikawa, and Ziyin Ma for their valuable feedback. This work is supported by NSF grant CNS-1814285.

## REFERENCES

[1] [n.d.]. . Retrieved September 19, 2020 from https://packages.gentoo.org/packages/net-libs/ngtcp2
[2] [n.d.]. *Chrome-har.* Retrieved January 18, 2021 from https://www.npmjs.com/package/chrome-har
[3] [n.d.]. *Implementations.* Retrieved September 18, 2020 from https://github.com/quicwg/base-drafts/wiki/Implementations
[4] [n.d.]. *Lighthouse.* Retrieved November 26, 2020 from https://developers.google.com/web/tools/lighthouse/
[5] [n.d.]. *netem.* Retrieved January 26, 2021 from https://wiki.linuxfoundation.org/networking/netem
[6] [n.d.]. *Puppeteer.* Retrieved January 18, 2021 from https://pptr.dev/
[7] 2019. *Speed Index.* Retrieved November 26, 2020 from https://web.dev/speed-index/
[8] 2020. . Retrieved October 18, 2020 from https://github.com/facebookincubator/mvfst/commit/8a98805d
[9] M. Allman, V. Paxson, and E. Blanton. 2009. *TCP Congestion Control.* RFC 5681. RFC Editor. http://www.rfc-editor.org/rfc/rfc5681.txt http://www.rfc-editor.org/rfc/rfc5681.txt.
[10] João Almeida. 2019. How fast is QUIC protocol and what makes Bolina faster - PT. II. https://blog.codavel.com/how-fast-is-quic-and-what-makes-bolina-faster-pt-ii
[11] Kornel Lesiński Andrew Galloni. 2019. *Parallel streaming of progressive images.* Retrieved November 29, 2020 from https://blog.cloudflare.com/parallel-streaming-of-progressive-images/
[12] P. Biswal and O. Gnawali. 2016. Does QUIC Make the Web Faster?. In *2016 IEEE Global Communications Conference (GLOBECOM).* 1–6.

[13] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, September-October (2016), 20 – 53. http://queue.acm.org/detail.cfm?id=3022184

[14] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. 2015. HTTP over UDP: An Experimental Investigation of QUIC. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (Salamanca, Spain) *(SAC '15)*. Association for Computing Machinery, New York, NY, USA, 609–614. https://doi.org/10.1145/2695664.2695706

[15] Yuchung Cheng, Neal Cardwell, and Nandita Dukkipati. 2017. *RACK: a time-based fast loss detection algorithm for TCP.* Internet-Draft draft-ietf-tcpm-rack-02. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-tcpm-rack-02.txt http://www.ietf.org/internet-drafts/draft-ietf-tcpm-rack-02.txt.

[16] S. Cook, B. Mathieu, P. Truong, and I. Hamchaoui. 2017. QUIC: Better for what and for whom?. In *2017 IEEE International Conference on Communications (ICC)*. 1–6.

[17] Ian Swett David Schinazi, Fan Yang. 2020. *Chrome is deploying HTTP/3 and IETF QUIC.* Retrieved October 8, 2020 from https://blog.chromium.org/2020/10/chrome-is-deploying-http3-and-ietf-quic.html

[18] Matt Menke Eric Roman. [n.d.]. *NetLog: Chrome's network logging system.* Retrieved January 18, 2021 from https://www.chromium.org/developers/design-documents/network-stack/netlog

[19] Uday Hiwarale. 2019. *How the browser renders a web page? — DOM, CSSOM, and Rendering.* Retrieved February 1, 2021 from https://medium.com/jspoint/how-the-browser-renders-a-web-page-dom-cssom-and-rendering-df10531c9969

[20] Jana Iyengar and Ian Swett. 2020. *QUIC Loss Detection and Congestion Control.* Internet-Draft draft-ietf-quic-recovery-29. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-quic-recovery-29.txt http://www.ietf.org/internet-drafts/draft-ietf-quic-recovery-29.txt.

[21] Jana Iyengar and Martin Thomson. 2020. *QUIC: A UDP-Based Multiplexed and Secure Transport.* Internet-Draft draft-ietf-quic-transport-32. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-32.txt http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-32.txt.

[22] Jana Iyengar and Martin Thomson. 2020. *QUIC: A UDP-Based Multiplexed and Secure Transport.* Internet-Draft draft-ietf-quic-transport-29. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-29.txt http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-29.txt.

[23] Subodh Iyengar. 2018. Moving Fast at Scale: Experience Deploying IETF QUIC at Facebook. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Heraklion, Greece) *(EPIQ'18)*. Association for Computing Machinery, New York, NY, USA, Keynote. https://doi.org/10.1145/3284850.3322434

[24] Lu Jun. 2020. *Simulate weak network environment using command line under Mac.* Retrieved January 26, 2021 from https://programmer.group/simulate-weak-network-environment-using-command-line-under-mac.html

[25] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. 2017. Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols. In *Proceedings of the 2017 Internet Measurement Conference* (London, United Kingdom) *(IMC '17)*. Association for Computing Machinery, New York, NY, USA, 290–303. https://doi.org/10.1145/3131365.3131368

[26] P. K. Kharat, A. Rege, A. Goel, and M. Kulkarni. 2018. QUIC Protocol Performance in Wireless Networks. In *2018 International Conference on Communication and Signal Processing (ICCSP)*. 0472–0476.

[27] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/3098822.3098842

[28] Jonathan Lipps. [n.d.]. *Simulating Different Network Conditions For Virtual Devices.* Retrieved November 26, 2020 from https://appiumpro.com/editions/104-simulating-different-network-conditions-for-virtual-devices

[29] Diego Madariaga, Lucas Torrealba, Javier Madariaga, Javiera Bermúdez, and Javier Bustos-Jiménez. 2020. Analyzing the Adoption of QUIC From a Mobile Development Perspective. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Virtual Event, USA) *(EPIQ '20)*. Association for Computing Machinery, New York, NY, USA, 35–41. https://doi.org/10.1145/3405796.3405830

[30] Robin Marx. 2020. *Will HTTP/3 really be faster than HTTP/2? Perhaps.* Retrieved October 10, 2020 from https://github.com/rmarx/holblocking-blogpost

[31] Robin Marx., Tom De Decker., Peter Quax., and Wim Lamotte. 2019. Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC. In *Proceedings of the 15th International Conference on Web Information Systems and Technologies - Volume 1: WEBIST,*. INSTICC, SciTePress, 130–143. https://doi.org/10.5220/0008191701300143

[32] Robin Marx, Joris Herbots, Wim Lamotte, and Peter Quax. 2020. Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Virtual Event, USA) *(EPIQ '20)*. Association for Computing Machinery, New York, NY, USA, 14–20. https://doi.org/10.1145/3405796.3405828

[33] Robin Marx, Maxime Piraux, Peter Quax, and Wim Lamotte. 2020. Debugging QUIC and HTTP/3 with Qlog and Qvis. In *Proceedings of the Applied Networking Research Workshop* (Virtual Event, Spain) *(ANRW '20)*. Association for Computing Machinery, New York, NY, USA, 58–66. https://doi.org/10.1145/3404868.3406663

[34] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. 1996. *TCP Selective Acknowledgment Options.* RFC 2018. RFC Editor.

[35] Yang Chi Matt Joras. 2020. *How Facebook is bringing QUIC to billions.* Retrieved November 12, 2020 from https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/

[36] Mattt. 2019. *Network Link Conditioner.* Retrieved November 13, 2020 from https://nshipster.com/network-link-conditioner/

[37] Patrick Meenan. 2019. *Better HTTP/2 Prioritization for a Faster Web.* Retrieved November 29, 2020 from https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web/

[38] Javad Nejati and Aruna Balasubramanian. 2020. WProfX: A Fine-Grained Visualization Tool for Web Page Loads. *Proc. ACM Hum.-Comput. Interact.* 4, EICS, Article 73 (June 2020), 22 pages. https://doi.org/10.1145/3394975

[39] Kyle Nekritz and Subodh Iyengar. 2017. *Building Zero protocol for fast, secure mobile connections.* Retrieved August 4, 2020 from https://engineering.fb.com/android/building-zero-protocol-for-fast-secure-mobile-connections/

[40] K. Nepomuceno, I. N. d. Oliveira, R. R. Aschoff, D. Bezerra, M. S. Ito, W. Melo, D. Sadok, and G. Szabó. 2018. QUIC and TCP: A Performance Evaluation. In *2018 IEEE Symposium on Computers and Communications (ISCC)*. 00045–00051.

[41] Minh Nguyen, Hadi Amirpour, Christian Timmerer, and Hermann Hellwagner. 2020. Scalable High Efficiency Video Coding Based HTTP Adaptive Streaming over QUIC. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Virtual Event, USA) *(EPIQ '20)*. Association for Computing Machinery, New York, NY, USA, 28–34. https://doi.org/10.1145/3405796.3405829

[42] Kazuho Oku and Jana Iyengar. 2020. *Can QUIC match TCP's computational efficiency?* Retrieved August 4, 2020 from https://www.fastly.com/blog/measuring-quic-vs-tcp-computational-efficiency

[43] Mohammad Rajiullah, Andra Lutu, Ali Safari Khatouni, Mah-Rukh Fida, Marco Mellia, Anna Brunstrom, Ozgu Alay, Stefan Alfredsson, and Vincenzo Mancuso. 2019. Web Experience in Mobile Networks: Lessons from Two Million Page Visits. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 1532–1543. https://doi.org/10.1145/3308558.3313606

[44] Jan Rüth, Konrad Wolsing, Klaus Wehrle, and Oliver Hohlfeld. 2019. Perceiving QUIC: Do Users Notice or Even Care?. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 144–150. https://doi.org/10.1145/3359989.3365416

[45] Darius Saif, Chung-Horng Lung, and Ashraf Matrawy. 2020. An Early Benchmark of Quality of Experience Between HTTP/2 and HTTP/3 using Lighthouse. arXiv:2004.01978 [cs.NI]

[46] Marten Seemann and Jana Iyengar. 2020. Automating QUIC Interoperability Testing. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Virtual Event, USA) *(EPIQ '20)*. Association for Computing Machinery, New York, NY, USA, 8–13. https://doi.org/10.1145/3405796.3405826

[47] Daniel Stenberg. 2019. *Comparison with HTTP/2.* Retrieved November 26, 2020 from https://http3-explained.haxx.se/en/h3/h3-h2

[48] Daniel Stenberg. 2020. *QUIC WITH WOLFSSL.* Retrieved September 19, 2020 from https://daniel.haxx.se/blog/2020/06/18/quic-with-wolfssl/

[49] Ian Swett. 2019. *From gQUIC to IETF QUIC and Beyond.* Retrieved February 4, 2021 from https://mile-high.video/files/mhv2019/pdf/day1/1_11_Swett.pdf

[50] Sreeni Tellakula. 2020. *Comparing HTTP/3 vs. HTTP/2 Performance.* Retrieved August 5, 2020 from https://blog.cloudflare.com/http-3-vs-http-2/

[51] Jack Wallen. 2018. *How to enable TCP BBR to improve network speed on Linux.* Retrieved October 11, 2020 from https://www.techrepublic.com/article/how-to-enable-tcp-bbr-to-improve-network-speed-on-linux/

[52] Alex Yu. 2020. *Benchmarking QUIC.* Retrieved September 15, 2020 from https://medium.com/@the.real.yushuf/benchmarking-quic-1fd043e944c7

[53] Y. Yu, M. Xu, and Y. Yang. 2017. When QUIC meets TCP: An experimental study. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*. 1–8.